



User Manual

Notice

This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2002-2004 Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Visual C# are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Table of Contents

User Manual	1
Notice	2
Table of Contents	iii
1. Introduction	1
2. Architecture	2
3. Using the Model Checker	3
3.1 Visual Studio integration	3
3.2 Using the Zing model viewer.....	3
3.2.1 Generating traces.....	3
3.2.2 Viewing traces and states	4
3.2.3 Viewing source code	4
3.2.4 Visualizing traces with Dot	4
3.3 Verifying a Model.....	5
3.4 Errors	5
3.4.1 Stuck states	5
3.4.2 Assertion failures.....	5
3.4.3 Execution failures.....	6
4. Checking refinement	7
4.1 Zing refinement relations	7
4.2 Using the refinement checker	7
4.3 Error messages	9
4.3.1 Simulation failure.....	10
4.3.2 Readiness failure	11
4.3.3 Refusals failure.....	11
5. Model Checker Internals	13
5.1 Efficiency techniques.....	13
5.1.1 State deltas.....	13
5.2 Fingerprints	13
5.3 User guided state-space reduction.....	13
5.3.1 Atomic blocks	13
5.3.2 Assume statements	13
5.4 Automatic state-space reduction	14
5.4.1 Symmetry reduction	14
5.4.2 Partial-order reduction.....	14
5.4.3 Summarization.	14
5.5 Zing States	14
5.5.1 Global data	14
5.5.2 Heap data.....	15
5.5.3 Processes & stacks	15
5.5.4 Comparing Zing states	15
5.5.5 Garbage Collection.....	15

1. Introduction

Concurrent programs are hard to develop and test. While writing concurrent programs, the programmer has to consider every possible interleaving of events among various processes. In spite of several decades of research and engineering experience, few people write robust concurrent programs. Concurrency related bugs (sometimes called “heisenbugs”) still surface only in stress-tests, and these bugs are very hard to reproduce, debug and fix. With the advent of efforts such as the **.Net** platform, we are enabling more programmers to write distributed and concurrent programs. Thus, problems associated with concurrency are only going to be more widespread.

A technique called “model checking” has proven to be surprisingly effective in the design and testing of concurrent programs. Model checkers work by systematically exploring all possible states of the concurrent program. Industrial software has such large number of states that it is infeasible for any systematic approach to cover all the reachable states. Our goal is the following: ***suppose we manage to represent a “model” from a program, where a model abstractly represents only a small amount of information about the program, then it is feasible to systematically explore the states of the model.***

The Zing project has three components: (1) a modeling language for expressing executable concurrent models of software, (2) a model checking infrastructure for exploring the state space of Zing models, and (3) support infrastructure for generating Zing models automatically from common programming languages like VB, C/C++, C#, and MSIL.

This document is a user manual for Zing’s model checking infrastructure.

2. Architecture

Zing is designed to have flexible software architecture. The architecture is designed to promote an efficient division of labor between MSR and product groups, and make it possible for MSR to experiment and innovate in the core state-space exploration technology while allowing the product groups to tackle those aspects of the problem that are domain-specific —namely extraction of **Zing** models from their source code, and visualization for showing the results from the **Zing** state explorer.

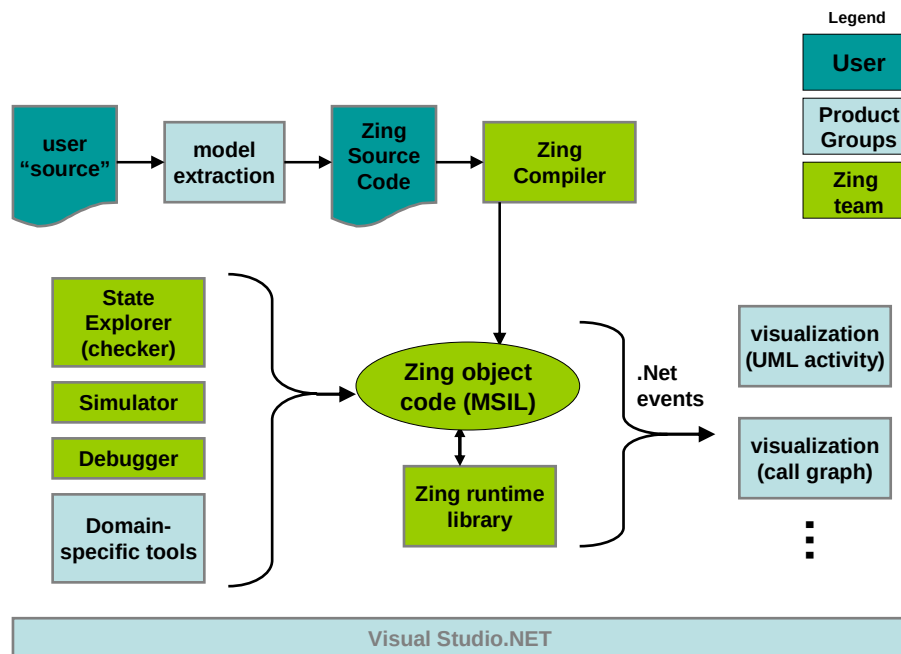


Figure 1: Architectural overview of Zing

We envision that clients with a need to do systematic state space exploration will be able to build a model extractor from their source language to **Zing** (to provide proof of concept, we are building some model extractors ourselves). Once model extraction is done, the generated **Zing** model is fed into a **Zing** compiler which converts the **Zing** model into MSIL object code. The object code supports a specific interface intended to be used by the systematic state explorer.

3. Using the Model Checker

Given a Zing model, the Zing compiler generates a managed assembly which allows the state-space of the model to be systematically explored. The state-space is explored by considering all possible interleavings of execution between concurrent processes as well as all possible outcomes of non-deterministic operations.

The Zing model viewer (Viewer.exe) can be used to manually examine the state space or to perform a full state-space exploration by the Zing model checker or the Zing refinement checker. The model-checker finds errors by detecting states in which:

- An explicitly programmed assertion is violated
- The model is stuck (i.e. no processes are executable, but some processes are not in a valid end state).
- An exception has been thrown by the Zing runtime (e.g. dereferencing a null pointer)
- An explicitly programmed exception was not handled

The refinement checker explores the relative behavior of two Zing models - an implementation and a specification - and looks for scenarios in which the implementation does not conform to the behavior described by the specification. The Zing refinement checker is described in more in Section 4.

3.1 Visual Studio integration

Zing currently supports a basic level of integration with Visual Studio. A Zing project in Visual Studio corresponds to a single Zing model, and compiles into a single managed assembly in the usual way. Zing projects may contain multiple source files. Zing source code may be edited in Visual Studio with support for token coloring, syntax checking, and code-sense. (Note: project dependencies are ignored by the Zing compiler.)

To launch the Zing model viewer from Visual Studio, select "Debug|Start Without Debugging" (or press Ctrl-F5) which will cause the currently-selected project to be loaded. [Two caveats here: 1) the Zing project will not be recompiled automatically, and 2) the currently-selected project is not necessarily the same as the "Startup" project.]

3.2 Using the Zing model viewer

The Zing model viewer is a tool for generating and viewing execution traces from Zing models. When the model viewer is first launched, you have the option (from the "File" menu) of either 1) opening a single Zing model (for model-checking), or 2) opening both a specification and implementation model (for refinement-checking). When a model (or pair of models) is selected, the assemblies are loaded. As a convenience, you can open a single Zing model by simply dragging its DLL into the main window of the model viewer.

3.2.1 Generating traces

With an opened model (or a model pair), you normally run the model-checker (or the refinement checker) by selecting the appropriate command from the "Tools" menu. If an error is detected, a trace window will be opened showing the sequence of process interleavings and non-deterministic outcomes that led to the failure.

If you elected to open a single Zing model, then there are two other ways of producing an execution trace. The "Generate random trace" command (in the Tools menu) produces a trace by making a random selection of

process (or non-deterministic choice) at each state transition. The trace ends when the model terminates successfully, becomes stuck, or when a fixed limit on the trace length is reached (see Tools/Options).

Another way to generate traces is to explore the state space manually. A state space browser window can be displayed by selecting "Show state-space browser" from the Tools menu. This shows a tree control in which each node corresponds to a Zing state. Nodes are labeled according to the process or choice number which their parent state executed to reach them. To generate a trace from the state-space browser, right-click on a node and select "Generate trace". This generates a trace corresponding to the execution from the initial state of the model to the selected node.

3.2.2 Viewing traces and states

A trace window is a high-level view of the execution from the initial state of a model to some state of interest - usually one in which an error was detected. The nodes in a trace window show the process or choice selection that led to the current state. If Zing events were generated during the transition to a state, they can be viewed by expanding the state node. This can be useful for getting a high-level overview of the model's execution.

Refinement traces are somewhat more complex because they show the simultaneous execution of two different models. In a refinement trace, the specification and implementation traces are shown side by side, and their traces are aligned precisely at the points at which the two models performed the same external communication. These points of alignment are displayed in a blue font. To maintain proper correlation between the two traces, they are always displayed in their fully-expanded form.

Once a trace has been generated, the F7 and F8 shortcuts can be used to move backward and forward in time, respectively. To see the complete detail for a state, right-click on a node in the trace and select "New state detail window". A state detail window gives a structured view of the full Zing state including overview information, processes and stacks, global variables, the heap, and any generated events. Select the tab corresponding to the information you wish to view. You may open as many state detail windows as you wish. As you move forward and backward in the trace window, all of the state detail windows will be refreshed with the data for the currently-selected state.

The "processes" tab in the state detail window is split into two sections - a list of processes and a tree view showing the stack of the selected process. The boundary between the two sections can be adjusted as desired for optimal viewing. The process list shows the process name, the method currently executing, and the status of the process (runnable, blocked, or terminated). The stack view shows the methods on the stack along with their parameters and local variables.

3.2.3 Viewing source code

To view the model's source code, open a state detail window, right-click on the process of interest, and select "View process source". This opens a window showing the next statement to be executed by the process. If the process is blocked, the statement will be highlighted in red, otherwise the current statement is shown in blue.

By default, source windows display the Zing source code for the model. If your model uses SourceContext attributes to correlate Zing statements with foreign source code, the model viewer will load and display the foreign source code in its source windows. To access this feature, the first runnable statement in the model must include a SourceContext attribute.

3.2.4 Visualizing traces with Dot

To better understand the execution of a model, the model viewer can generate an interaction diagram showing the processes in a model, when they were created, the way in which their execution was interleaved, and the transfer of messages between them. To enable this feature, you must first install the third-party tools Graphviz and WinGraphviz from [here](#).

Whenever a trace window is displayed, an interaction diagram can be created by selecting "Generate interaction diagram" from the View menu. In an interaction diagram, the timeline for each process is shown in black. Solid segments indicate periods in which the process was executing; dashed lines are used when some other process was active. Green lines indicate the creation of one process by another, and blue lines show communication between processes. A black circle at the end of a timeline indicates normal completion of the process. A red circle denotes an error in the process.

When an interaction diagram is displayed, you can move the cursor over a state to cause the corresponding state in the trace to be selected (and all of the other detail windows to be updated).

Interaction diagrams can also be generated for refinement traces. These look much like the diagrams described above with the following additions:

- The specification and implementation diagrams are shown with a green and a beige background, respectively.
- External communication events are displayed in the timeline for each model.
- The correlation of communication events between the two models is indicated by lines connecting the corresponding events.

3.3 Verifying a Model

To perform a full exploration of the state space of a model, select Verify/Verify! (ctrl-V). By default, the verification will stop when the first error is found. To continue searching after errors are found, go to the "Stop on first error" item in the "Verify" menu and un-check it.

When the verifier finds an error state, it expands the state tree as necessary to show the error state. Error states are always highlighted in light red in the state tree. Normal end-states are shown in blue. States in which an "assume" statement has failed are shown in yellow. When an error state is found and selected, the F7 and F8 shortcuts may be used to move the state tree's node selection backward and forward in time, respectively. Combined with the source tracking windows, this can be a very convenient way of walking through the sequence of events which led to an error.

3.4 Errors

The Zing model-checker is capable of detecting a number of different errors in a Zing model, as described in the sections below. All properties currently checked are safety properties. Liveness properties are not currently supported.

3.4.1 Stuck states

Zing will detect states in which no further progress can be made but are not valid "end states". A valid end state is one in which each process has terminated normally (by returning from its top-level method) or is blocked in a select statement marked with the end qualifier.

3.4.2 Assertion failures

The `assert` statement allows application-specific safety properties to be encoded in a Zing model. If the asserted expression does not evaluate to `true`, then the assertion fails and an error trace is reported. An optional comment string may be included in the `assert` statement to provide additional domain-specific context for the error.

3.4.3 Execution failures

Zing models may fail to execute properly for several other reasons. Any of these failures will cause an error trace to be generated by the model-checker:

- Dereferencing a null pointer
- Assigning a generic “object” reference to a strongly-typed variable of an incompatible type
- Indexing an array outside of its declared bounds
- Arithmetic overflow
- Divide by zero
- Failure to handle a Zing exception
- Removing a non-existent member of a set
- Executing a blocking `select` statement within the body of an atomic block (i.e. not as the first statement of the block)
- Execution of a `choose` operator with zero alternatives (e.g. choose from an empty set)

4. Checking refinement

4.1 Zing refinement relations

The Zing refinement checker currently supports three distinct refinement relations:

- Simulation
- Stable failures refinement
- Stuck-free conformance

These relations are defined over external, visible events that can be generated by the Zing `event` statement together with a special internal, invisible event. The latter event is usually referred to as τ . External events are used, e.g., to model send and receive actions of a message-passing component that can be observed from the outside. The internal, invisible event τ is automatically generated from computation steps in the model checker other than those generated from executing `event` statements.

The refinement relations of Zing and the Zing refinement checker are based on the semantic theory of CCS^{1 2} and uses ideas from the theory of refinement in CSP^{3 4}. The three refinement relations of Zing form a hierarchy, where simulation is the coarsest (largest) relation and stuck-free conformance is the finest (smallest) relation.

Intuitively, simulation between an implementation and a specification requires that, whenever an event can be generated by the implementation, then the same event can also be generated by the specification. An important use of simulation is to specify an implementation process by a more nondeterministic process. A precise definition of simulation and its theory can be found in [1,2].

Stable failures refinement is a restriction of simulation that is useful for reasoning about deadlock (i.e., states where a process is blocked forever waiting for an event that will never occur). A state of a model is called stable, if no internal (τ) event can be generated from the state. In particular, such a state has no internal nondeterminism since all of Zing's constructs for nondeterministic internal choice generate τ -events. An implementation model refines a specification model under stable failures refinement if simulation holds between the models and, in addition, every future stable state in the implementation can be matched by one in the specification such that the matching specification state does not generate any events that can not be generated by the implementation state. Another way of putting this is to say that, whenever the implementation can *fail* to generate a set of events from a stable state, the specification can also fail to generate at least that set of events from some stable state. The CSP theory of stable failures refinement can be found in [4]. The papers on stuck-free conformance^{5 6 7} contain information about how stable failures refinement is adapted into CCS and Zing.

Stuck-free conformance is a restriction of stable failures refinement that is useful for reasoning about both deadlock and unreceived messages in asynchronous systems. Stuck-free conformance restricts the stable failures relation by requiring that, if a state in the implementation has a future stable state that fails to generate events X while (at the same time) being ready to generate event a , then the the specification can do the same from some future stable state. The papers [5,6] describe the theory of stuck-free conformance in detail, and the paper [7] describes how it is implemented in the Zing model checker and used on some examples.

4.2 Using the refinement checker

The Zing refinement checker can be invoked from the Zing model viewer (Viewer.exe). The refinement checker is always invoked on two separate dll files containing the models that are to be compared for refinement. The dll files must first be generated by the Zing compiler (zc.exe). Each dll file must contain a `static activate`

method, since each dll file must define a runnable process. The refinement checker is invoked from the model viewer by selecting File/Open Spec & Impl models, and then selecting the two dlls representing the implementation and the specification. Here, the words “implementation” and “specification” simply mean that the Zing refinement checker will check that the “implementation” refines (simulates, failure-refines, or conforms to) the “specification”. After selecting the two models, one can select Tools/Options/Refinement Checker and then select a desired refinement relation by checking one of the Conformance, Simulation and Stable failures selections.

For example, suppose that file1.zing contains the code

```
class tst1{
    static activate void Run1(){
        event(0, 1, true);
    }
};
```

and file2.zing contains the code

```
class tst2{
    static activate void Run2(){
        select visible{
            wait(true) -> event(0, 1, true);
            wait(true) -> event(0, 2, true);
        }
    }
};
```

then we can compile file1.zing to file1.dll and file2.zing to file2.dll and then invoke the refinement checker on file1.dll and file2.dll.

Zing contains a number of constructs that are of special significance in refinement checking and that can be used to translate CCS [1,2] into Zing. For example, the CCS process

$$P = a + b$$

can be modeled by the Zing program

```
class Action{
    static int a = 0;
    static int b = 1;
};
class P{
    static activate void Run(){
        select visible{
            event(0, Action.a, false);
            event(0, Action.b, false);
        }
    }
};
```

Here, we have used the (arbitrary) convention that `event(0, Action.a, false)` represents action a. The `select visible` statement must be used to faithfully model all the variations of CCS choice, including the mixed choice:

$$P = a + \tau.b$$

which can be written in Zing as:

```
class Action{
    static int a = 0;
    static int b = 1;
};
class P{
    static activate void Run(){
        select visible{
            event(0, Action.a, false);
            wait(true) -> event(0, Action.b, false);
        }
    }
};
```

Here, the condition `wait(true)` will generate a τ -event, which can be observed by the Zing refinement checker. Internal nondeterministic choice, as in

$$P = \tau.a + \tau.b$$

can be written as

```
class Action{
    static int a = 0;
    static int b = 1;
};
class P{
    static activate void Run(){
        select visible{
            wait(true) -> event(0, Action.a, false);
            wait(true) -> event(0, Action.b, false);
        }
    }
};
```

Parallel composition is expressed by `async` calls to methods containing the code for each component.

4.3 Error messages

A refinement check can fail for four different kinds of reasons. First, since a refinement check implies a model check of both implementation and specification, any error condition of the model checker found in either the implementation model or the specification model (section 3.4) will be reported as part of a refinement check. Such errors are called *implementation failures* and *specification failures*, respectively, in the Zing Viewer.

The remaining three kinds of errors cover failure of one of the refinement relations. The relations are composed of three requirements, namely a *simulation* condition (the implementation must be simulated by the specification), a *refusals* condition (the specification must refuse all events refused by the implementation), and, in the case of stuck-free conformance, a *readiness* condition that is coupled with the refusals condition (whenever the implementation can refuse a set of events and generate an event from a stable state then so can the specification).

4.3.1 Simulation failure

A simulation failure occurs when the implementation can generate an event that cannot be matched by the specification. For example, suppose that we have the shared action type

```
class Action{
    static int a = 0;
    static int b = 1;
    static int c = 2;
};
```

and an implementation and a specification defined by the classes, respectively:

```
class Impl{
    static activate void Run(){
        event(0, Action.a, true);
        select visible{
            wait(true) -> event(0, Action.b, true);
            wait(true) -> event(0, Action.c, true);
        }
    }
};
```

and

```
class Impl{
    static activate void Run(){
        event(0, Action.a, true);
        select visible{
            wait(true) -> event(0, Action.b, true);
        }
    }
};
```

Here, the refinement checker will report a Simulation Failure, because the specification cannot perform the action c. The Zing Viewer shows the event that cannot be matched by the specification, in this case the event `event(0, Action.c, true)`

which is indicated as “0/2/!” in the viewer, where “!” is used as shorthand for the value `true` and “?” is used for the value `false`. In addition to showing the event, the viewer also shows two error traces, one in the

Implementation window and one in the Specification window. The trace shows a sequence of steps from the initial states of both processes up until the point where the event occurs that witnesses failure of simulation.

4.3.2 Readiness failure

A readiness failure occurs when the implementation is in a stable state and can generate a certain event, but the specification cannot go (on internal transitions) to a stable state where it can generate the same event.

For example, suppose we have (assuming the class `Action` above):

```
class Impl{
    static activate void Run(){
        select visible{
            event(0, Action.a, false) -> ;
        }
    }
};
```

and

```
class Impl{
    static activate void Run(){
        select visible{
            wait(true) -> ;
            event(0, Action.a, false) -> ;
        }
    }
};
```

The refinement checker reports a Readiness Failure, because the implementation can generate the event 0/0? From a stable state, but the specification cannot do so -- the state from which 0/0? can be generated is unstable due to the choice `wait(true)`, which generates an internal (τ) event. The set of actions that the implementation is able to generate from the offending stable state is shown in a window called Implementation Ready Set.

4.3.3 Refusals failure

A refusals failure occurs when the implementation is in a stable state and refuses a set of events, but the specification cannot go (on internal transitions) to a stable state from which that set (or a superset thereof) is refused.

For example, if we turn the previous example around, we have:

```
class Impl{
    static activate void Run(){
        select visible{
            wait(true) -> ;
            event(0, Action.a, false) -> ;
        }
    }
};
```

```
};  
and  
class Spec{  
    static activate void Run(){  
        select visible{  
            event(0, Action.a, false) -> ;  
        }  
    }  
};
```

The refinement checker reports a Refusals Failure, because the implementation can go (on internal transitions) to a stable state from which the event 0/0? is refused (by choosing the condition `wait(true)`), but the specification cannot do the same. In addition to showing the event refused by the implementation (called the Refusal event in the Zing Viewer), error traces from the implementation and specification are shown in the Implementation and Specification windows, respectively, showing the transitions up until the point where the offending refusal event is found.

5. Model Checker Internals

The Zing compiler and runtime are designed to explore the state-space of a model efficiently using a number of different techniques. In some cases we strive for efficiency by making each state consume less space, be faster to generate, or be more quickly compared with other states. Usually more important, though, are techniques that attempt to reduce the size of the state-space itself.

Some of the techniques currently employed by Zing are described here. We expect additional techniques to be applied in the future.

5.1 Efficiency techniques

5.1.1 State deltas

It's common in a Zing model for one step of execution to make a relatively small change to the overall state of the model. The Zing runtime uses this to its advantage by remembering only the “delta” between a state and its parent. As the state-space search proceeds (in a depth-first manner, typically) only the “current” state is maintained in its complete form. When it becomes necessary to return to the parent of a state (to generate its other successors) the reverse-delta that was maintained for it can be quickly applied, yielding the parent's full state representation.

State deltas result in a considerable savings of both time and space. But because it is also possible to generate the full “clone” of any Zing state, we retain the ability to implement more complex search strategies in the future.

5.2 Fingerprints

Because the state of a Zing model is complex, the comparison of one Zing state with another presents a challenge. To address this, Zing computes a 64-bit fingerprint for each state which uniquely identifies the state (with high probability). This allows states to be hashed and compared efficiently. As an additional benefit of the fingerprinting algorithm, “similar” states yield the same fingerprint, reducing the number of states that must be considered (see Section 5.4.1 below).

5.3 User guided state-space reduction

The user has two ways to guide state-space reduction.

5.3.1 Atomic blocks

Atomic blocks allow the author of a Zing model to control the points at which interleavings of execution between processes will be considered. In many cases, atomic blocks must be used to achieve the desired model semantics – e.g. to ensure that operations which are truly atomic are treated as such by Zing. In other cases, additional atomic blocks can be added to a model without affecting its semantics but with the intent of reducing the number of process interleavings to be considered.

5.3.2 Assume statements

The assume statement allows Zing authors to manually “prune” portions of the state-space tree. The assume statement includes a binary expression whose result determines whether the current state is worthy of

consideration. If the expression evaluates to `false`, then the current state is deemed “uninteresting” and neither it nor its successors will be examined further.

This is similar to the `assert` statement in that it may remove states from consideration, but unlike an assertion failure it is not treated as an error.

5.4 Automatic state-space reduction

We have implemented several algorithms to automatically reduce the number of states explored, without missing any errors that may exist in the model. We give a brief overview of the techniques below. More details can be obtained from the technical papers at the Zing website <http://research.microsoft.com/zing>

5.4.1 Symmetry reduction

A Zing state comprises the thread stacks, the global variables, and a heap of dynamically allocated objects. Two states are equivalent if the contents of the thread stacks and global variables are identical and the heaps are isomorphic. When the state explorer discovers a new state, it first constructs a canonical representation of the state by traversing the heap in a deterministic order. It then stores a fingerprint of this canonical representation in the hash table.

5.4.2 Partial-order reduction.

We have implemented a state-reduction algorithm that has the potential to reduce the number of explored states exponentially without missing errors. This algorithm is based on Lipton's theory of reduction. Our algorithm is based on the insight that in well-synchronized programs, any computation of a thread can be viewed as a sequence of transactions, each of which appears to execute atomically to other threads. During state exploration, it is sufficient to schedule threads only at transaction boundaries. If programmers follow the discipline of protecting each shared variable with a lock, then these transactions can be inferred automatically. These inferred transactions reduce the number of interleavings to be explored, and thereby greatly alleviate the problem of state explosion.

5.4.3 Summarization.

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. This is not the case for concurrent programs. Zing has an implementation of a novel model checking algorithm for concurrent programs that uses procedure summarization as an essential component. Our method for procedure summarization is based on the insight about transactions mentioned earlier. We summarize within each transaction; the summary of a procedure comprises the summaries of all transactions within the procedure. The procedure summaries computed by our algorithm allow reuse of analysis results across different call sites in a concurrent program, a benefit that has hitherto been available only to sequential programs.

5.5 Zing States

Unlike many model-checkers, Zing's state vector size is variable in a number of aspects. The elements of a Zing state, and the way in which states are compared are described below.

5.5.1 Global data

The “global data” portion of a Zing state is comprised of the static fields of all of the declared classes in the Zing model. The size of the global data section is constant for a given model.

5.5.2 Heap data

The Zing heap contains all of the complex type instances allocated by the model. The heap size is variable in both the number of items it contains, and (for some types) the size of the items themselves. Channels, sets, and (eventually) arrays are complex types whose size is variable.

5.5.3 Processes & stacks

Zing states may include an unbounded number of processes, and each process includes a stack whose size is unbounded. The size of an individual stack frame is fixed.

5.5.4 Comparing Zing states

Zing states are compared against one another not by direct comparison of their state data, but rather by comparing their *fingerprints*. The 64-bit fingerprint of a state is generated by first serializing the state data in a prescribed manner and then applying a fingerprinting algorithm to the generated state vector. The fingerprinting algorithm is chosen both for efficiency and to minimize the possibility that two different states will yield the same fingerprint.

Another property of the fingerprinting process is that it will produce the same fingerprint for states that are superficially different, but actually equivalent in some respect. This is done primarily by “normalizing” pointers to the Zing heap during the process of serializing the Zing state.

5.5.5 Garbage Collection

As part of computing each state’s fingerprint, any live objects in the Zing heap will be encountered and serialized. Any unreachable heap objects are simply not carried forward to subsequent states.

¹ Robin Milner: *Communication and Concurrency*. Prentice Hall 1989.

² Robin Milner: *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press 1989.

³ C. A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall 1985.

⁴ A. W. Roscoe: *The Theory and Practice of Concurrency*. Prentice Hall 1998.

⁵ C. Fournet, C. A. R. Hoare, S. K. Rajamani, J. Rehof: *Stuck-free Conformance*. Proceedings CAV 2004, Computer Aided Verification. Springer Lecture Notes in Computer Science.

⁶ C. Fournet, C. A. R. Hoare, S. K. Rajamani, J. Rehof: *Stuck-free Conformance Theory for CCS*. Microsoft Research Technical Report MSR-TR-2004-09, February 2004.

⁷ T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, Y. Xie: *Zing: Exploiting Program Structure for Model Checking Concurrent Software*. Invited paper, CONCUR 2004.